A Deep Dive into the Interprocedural Optimization Infrastructure

Outline

- What is IPO? Why is it?
- Introduction of IPO passes in LLVM
- Inlining
- Attributor

What is IPO?

What is IPO?

- Pass Kind in LLVM
  - Immutable pass
  - Loop pass
  - Function pass

  Intraprocedural
  - Call graph SCC pass
  - Module pass

  Interprocedural

IPO considers more than one function at a time

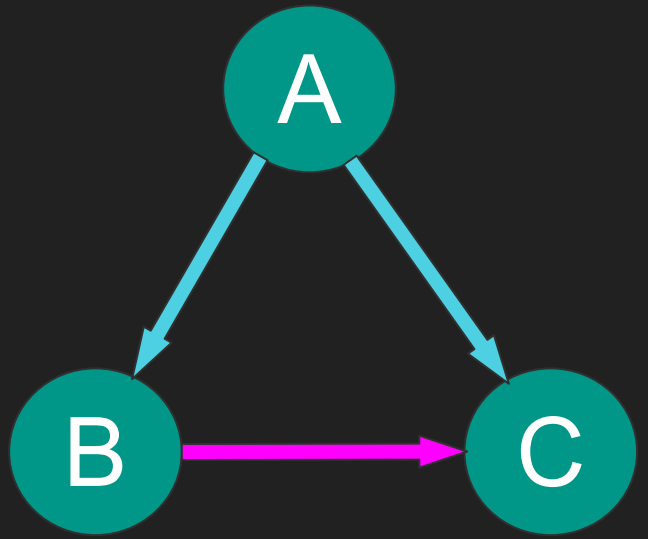
Categorize IPO passes

- Propagation between caller and callee
  - InferFunctionAttrs, ArgumentPromotion, DeadArgumentElimination, etc.
  - Two directions
    - Top-bottom, bottom-up, or both
- Inliner
  - Inliner, SimpleInliner, AlwaysInliner, etc.
- Linkage and Globals related
  - GlobalDCE, GlobalOpt, ConstantMerge, etc.
- Others
  - MergeFunction, OpenMPOpt, HotColdSplitting, etc

Call Graph

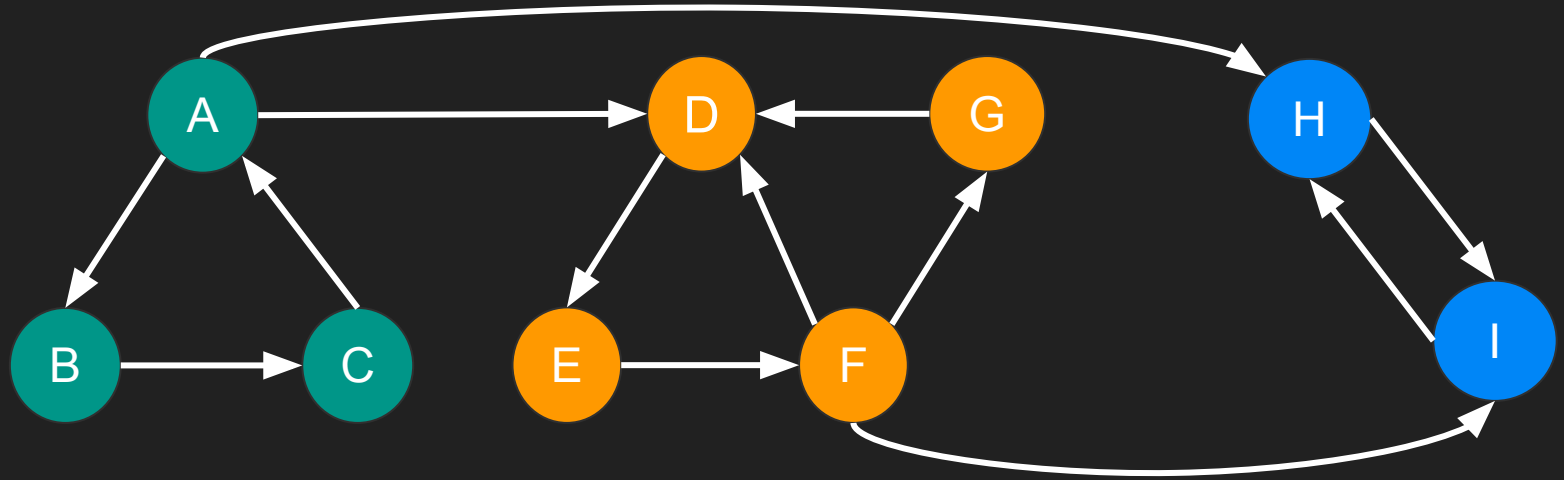
- Node : functions
- Edge : from caller to callee

```
void A() {
  B();
  C();
}
void B() {
  C();
}
void C() {
  ...
}
```



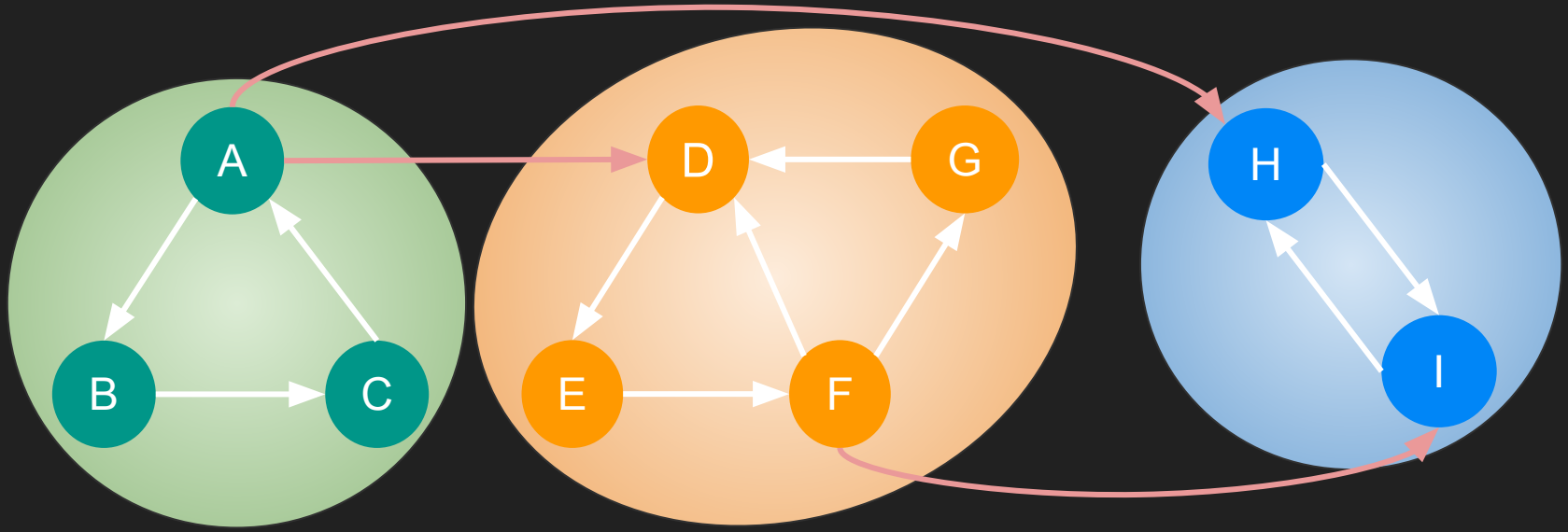
Call Graph SCC

- SCC stands for “Strongly Connected Component”



Call Graph SCC

- SCC stands for “Strongly Connected Component”



Passes In LLVM

IPO passes in LLVM

- Where
  - Almost all IPO passes are under `llvm/lib/Transforms/IPO`

Categorization of IPO passes

- Inliner
  - AlwaysInliner, Inliner, InlineAdvisor, ...
- Propagation between caller and callee
  - Attributor, IP-SCCP, InferFunctionAttrs, ArgumentPromotion, DeadArgumentElimination, ...
- Linkage and Globals
  - GlobalDCE, GlobalOpt, GlobalSplit, ConstantMerge, ...
- Others
  - MergeFunction, OpenMPOpt, HotColdSplitting, Devirtualization...

Why is IPO?

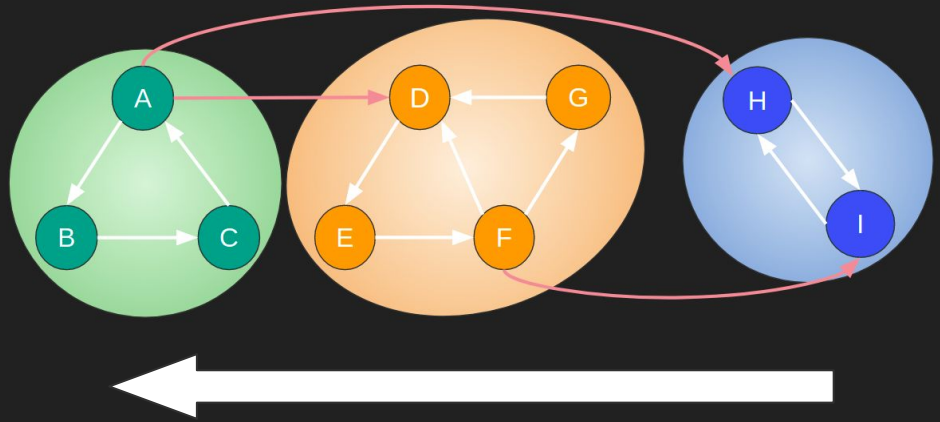
- Inliner
  - Specialize the function with call site arguments
  - Expose local optimization opportunities
  - Save jumps, register stores/loads (calling convention)
  - Improve instruction locality
- Propagation between caller and callee
  - Other passes would benefit from the propagated information
- Linkage and Globals related
  - Exploit the fact all uses of internal values are known
  - Remove unused internal globals
  - Cooperates with LTO

Pass Kind

- Module Pass[1]
  - Take a module as a “unit”
  - The most coarse-grained pass kind

Pass Kind

- Call Graph SCC Pass[1]
  - Take a SCC of call graph as a “unit”
  - Applied in post order of call graph
    - bottom-up
- Allowed
  - Modify the current SCC
  - Add or remove globals
- Disallowed
  - Modify any SCCs other than the current one
  - Add or remove SCC



Common IPO Pitfalls

- Scalability
- Complicated linkages
- Optimization pipeline, phase ordering
- Function pointer, different “kinds” of call sites, non-call site uses, …
- Variadic functions, complicated attributes (naked, byval, inreg, …)
- Keeping call graphs updated (for new and old pass managers)
  - `CallGraph` … old PM
  - `LazyCallGraph` … new PM

Existing IPO passes

Pass introduction style template -<option>

- [What]
  - [How]
  - [Notes]
  - ...

```
; Example
; https://romannurik.github.io/SlidesCodeHighlighter/
; Theme: Dark
; Font size: 9pt
define i32 before() {
   ...
}
```

```
define i32 after() {
   ...
}
```

Simple inliner `-inline`

- Bottom-up Inlining
  - CGSCC pass
- Example

```
void foo(int cond) {
  if (cond) {
    /* hot */
    ...
  } else {
    /* cold */
    ...
  }
}

void use_foo() {
    foo(x);
}
```

```
void use_foo() {
  if (x) {
    /* hot */
    ...
  } else {
    /* cold */
    ...
  }
}
```

Partial inliner `-partial-inliner`

- Inlining hot region only
- Example

```
void foo(int cond) {
  if (cond) {
    /* hot */
    ...
  } else {
    /* cold */
    ...
  }
}

void use_foo() {
    foo(x);
}
```

```
void foo.cold() {
  /* cold */
  ...
}

void use_foo() {
  if (x) {
    /* hot */
    ...
  } else {
    foo.cold();
  }
}
```

Always inliner `-always-inline`

- Try to inline functions marked “`alwaysinline`”
- Runs even in `-O0` or with llvm passes disabled!
- Basically overrides the inliner heuristic.
- Example

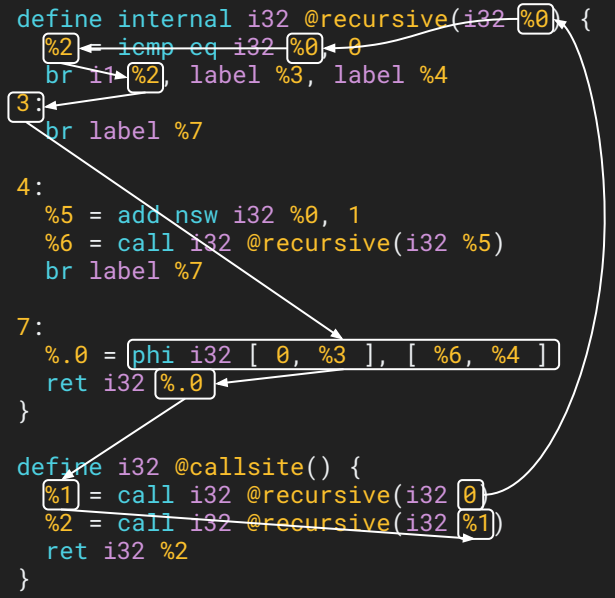
```
> cat test.ll

define i32 @inner() alwaysinline {
entry:
  ret i32 1
}

define i32 @outer() {
entry:
  %ret = call i32 @inner()
  ret i32 %ret
}
```

```
> opt -always-inline test.ll -S

define i32 @inner() alwaysinline {
entry:
  ret i32 1
}

define i32 @outer() {
entry:
  ret i32 1
}
```

IPSCCP `-ipsccp`

- Interprocedural Sparse Conditional Constant Propagation
- Blocks and instructions are assumed dead until proven otherwise.
- Traverses the IR to see which Instructions/Blocks/Functions are alive and which values are constant.

IPSCCP: Example



```
define internal i32 @recursive(i32 %0) {
  %2 = icmp eq i32 %0, 0
  br i1 %2, label %3, label %4

3:
  br label %7

4:
  %5 = add nsw i32 %0, 1
  %6 = call i32 @recursive(i32 %5)
  br label %7

7:
  %.0 = phi i32 [ 0, %3 ], [ %6, %4 ]
  ret i32 %.0
}

define i32 @callsite() {
  %1 = call i32 @recursive(i32 0)
  %2 = call i32 @recursive(i32 %1)
  ret i32 %2
}
```

```
define internal i32 @recursive(i32 %0) {
  br label %2

2:
  br label %3

3:
  ret i32 undef
}

define i32 @callsite() {
  %1 = call i32 @recursive(i32 0)
  %2 = call i32 @recursive(i32 0)
  ret i32 0
}
```

Argument Promotion `-argpromotion`

- Promote “by pointer” arguments to be “by value” arguments
  - If the argument is only “loaded”
  - Handle both `load` and `GEP` instructions
  - Pass the loaded value to the function, instead of the pointer
- Flow
  - Save information about loads of viable arguments
  - Create new function
  - Insert such load instructions to the caller
- This is (partially) subsumed by the Attributor

Argument Promotion: Example

```
> cat test.ll

%T = type { i32, i32 }
@G = constant %T { i32 17, i32 0 }

define internal i32 @test(%T* %p) {
entry:
  %a.gep = getelementptr %T, %T* %p, i64 0, i32 0
  %a = load i32, i32* %a.gep
  %v = add i32 %a, 1
  ret i32 %v
}

define i32 @caller() {
entry:
  %v = call i32 @test(%T* @G)
  ret i32 %v
}
```

```
> opt -S -argpromotion test.ll

%T = type { i32, i32 }
@G = constant %T { i32 17, i32 0 }

define internal i32 @test(i32 %p.0.0.val) {
entry:
  %v = add i32 %p.0.0.val, 1
  ret i32 %v
}

define i32 @caller() {
entry:
  %G.idx = getelementptr %T, %T* @G, i64 0, i32 0
  %G.idx.val = load i32, i32* %G.idx
  %v = call i32 @test(i32 %G.idx.val)
  ret i32 %v
}
```

InferFunctionAttrs `-inferattrs`

- Annotate function attrs on known library functions.
- Example

```
> cat test.ll

define i8* @foo() {
  %1 = call i8* @malloc(i64 1)
  ret i8* %1
}

declare i8* @malloc(i64)
```

```
> opt -inferattrs test.ll -S

define i8* @foo() {
  %1 = call i8* @malloc(i64 1)
  ret i8* %1
}

; Function Attrs: nofree nounwind
declare noalias i8* @malloc(i64) #0

attributes #0 = { nofree nounwind }
```

DeadArgumentElimination `-deadargelim`

- Remove dead arguments from internal functions
- How:
  - Delete arglist (...) if no `va_start` is called
  - Assume all arguments dead unless proven otherwise
- Example

```
; Dead arg only used by dead retval
define internal i32 @test(i32 %DEADARG) {
  ret i32 %DEADARG
}

define i32 @test2(i32 %A) {
  %DEAD = call i32 @test(i32 %A)  ; 0 uses
  ret i32 123
}
```

```
define internal void @test() {
  ret void  ; Argument was eliminated
}

define i32 @test2(i32 %A) {
  call void @test()
  ret i32 123
}
```

CalledValuePropagation

`-called-value-propagation`

- Add metadata to indirect call sites indicating potential callees
- Example

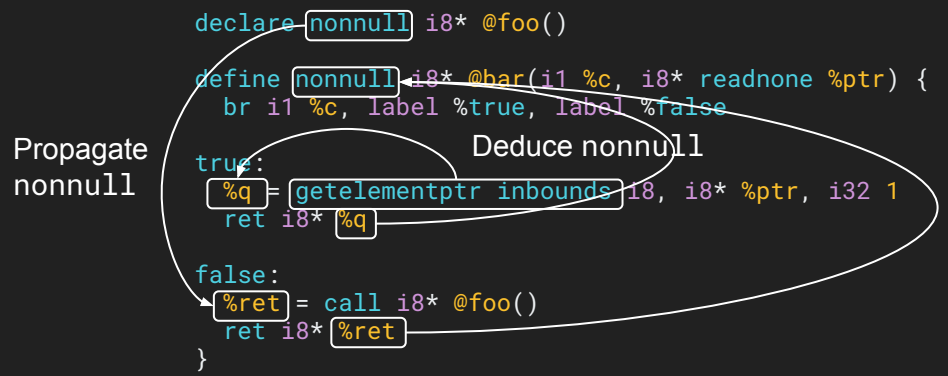
```
define void @test_select_entry(i1 %flag) {
entry:
    call void @test_select(i1 %flag)
    ret void
}

define internal void @test_select(i1 %f) {
entry:
    %tmp = select i1 %f, void ()* @foo_1, void ()* @foo_2
    call void %tmp()
    ret void
}

declare void @foo_1() norecurse
declare void @foo_2() norecurse
```

```
define void @test_select_entry(i1 %flag) {
entry:
  call void @test_select(i1 %flag)
  ret void
}

define internal void @test_select(i1 %f) {
entry:
  %tmp = select i1 %f, void ()* @foo_1, void ()* @foo_2
  call void %tmp0(), !callees !0
  ret void
}
declare void @foo_1() norecurse
declare void @foo_2() norecurse
!0 = !{void ()* @foo_1, void ()* @foo_2}
```

FunctionAttrs `-function-attrs -rpo-function-attrs`

- Deduce and propagate attributes
- Two versions
  - Bottom-up
  - Top-bottom (reverse post order)
- This is subsumed by the Attributor
- Example

```
declare nonnull i8* @foo()

define i8* @bar(i1 %c, i8* %ptr) {
  br i1 %c, label %true, label %false

true:
  %q = getelementptr inbounds i8, i8* %ptr, i32 1
  ret i8* %q

false:
  %ret = call i8* @foo()
  ret i8* %ret
}
```



PruneEH `-prune-eh`

- Remove unused exception handling code
  - Turn `invoke` into `call` when the callee is proven not to throw an exception
- Example

```
define void @foo() nounwind {
  ...
  ret void
}

define i32 @caller() personality i32 (...)* @eh_function {
  invoke void @foo( ) to label %Normal unwind label %Except


Normal:
  ret i32 0

Except:
  landingpad { i8*, i32 } catch i8* null
  ret i32 1
}
```

```
define void @foo() nounwind {
  ...
  ret void
}

define i32 @caller() #0 personality i32 (...)* @eh_function {
  call void @foo()  ; Note there's no invoke
  br label %Normal  ; and the %Except block was removed.

Normal:
  ret i32 0
}
```

https://llvm.org/docs/Passes.html#prune-eh-remove-unused-exception-handling-info

GlobalDCE `-globaldce`

- Eliminate unreachable internal globals
- An aggressive algorithm
  - Initially assume all globals are dead
- Example

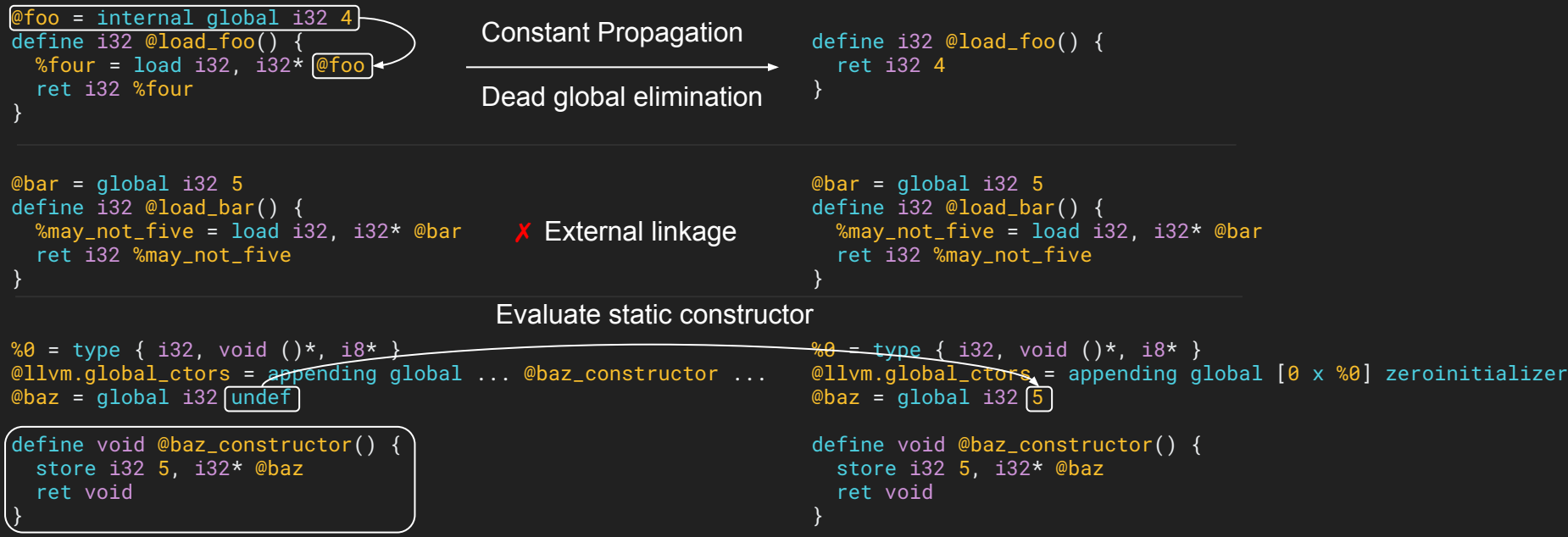
```
@A = global i32 0
@D = internal alias i32, i32* @A
@L1 = alias i32, i32* @A
@L2 = internal alias i32, i32* @L1
@L3 = alias i32, i32* @L2
```

```
@A = global i32 0

@L1 = alias i32, i32* @A
@L2 = internal alias i32, i32* @L1
@L3 = alias i32, i32* @L2
```

GlobalOpt `-globalopt`

- Optimize global values
  - Evaluate static constructors (`llvm.global_ctors`)
  - Optimize non-address-taken globals
    - Constant Propagation
    - Dead global elimination

GlobalOpt : Example



Constant Merge `-constmerge`

- Merge duplicate global constants together into a shared one
  - Construct a map from constants to globals
- Example

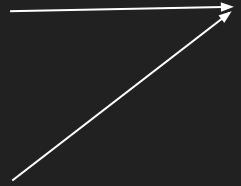
```
@foo = constant i32 6
@bar = internal unnamed_addr constant i32 6
@baz = constant i32 6

define i32 @use_bar(i32 %arg) {
  %six = load i32, i32* @bar
  %ret = add i32 %arg, %six
  ret i32 %ret
}
```

```
@foo = constant i32 6

@baz = constant i32 6

define i32 @use_bar(i32 %arg) {
  %six = load i32, i32* @foo, align 4
  %ret = add i32 %arg, %six
  ret i32 %ret
}
```

MergeFunctions `-mergefunc`

- Find equivalent functions and merge them
  - Introduce a “total order” among functions
  - Use binary search to find an equivalent function

```
define internal i64 @foo(i32* %P, i32* %Q) {
  store i32 4, i32* %P
  store i32 6, i32* %Q
  ret i64 0
}

define internal i64* @bar(i32* %P, i32* %Q) {
  store i32 4, i32* %P
  store i32 6, i32* %Q
  ret i64* null
}

define i64 @use_foo(i32* %P, i32* %Q) {
  %ret = call i64 @foo(i32* %P, i32* %Q)
  ret i64 %ret
}

define i64* @use_bar(i32* %P, i32* %Q) {
  %ret = call i64* @bar(i32* %P, i32* %Q)
  ret i64* %ret
}
```



```
define internal i64* @bar(i32* %P, i32* %Q) {
  store i32 4, i32* %P, align 4
  store i32 6, i32* %Q, align 4
  ret i64* null
}

define i64 @use_foo(i32* %P, i32* %Q) {
  %ret = call i64 bitcast (i64* (i32*, i32*)* @bar to
i64 (i32*, i32*)*)(i32* %P, i32* %Q)
  ret i64 %ret
}

define i64* @use_bar(i32* %P, i32* %Q) {
  %ret = call i64* @bar(i32* %P, i32* %Q)
  ret i64* %ret
}
```

https://llvm.org/docs/MergeFunctions.html

OpenMPOpt `-openmp-opt`

- *Various* OpenMP specific optimization
  - Runtime call deduplication
  - runtime call replacement
  - parallel region merging
  - GPU code optimization, …
- Example

```
; Runtime call deduplication

define void @test() {
  %nthreds1 = call i32 @omp_get_num_threads()
  call void @use(%nthreads1)
  %nthreds2 = call i32 @omp_get_num_threads()
  call void @use(%nthreads2)
  ret void
}
```

```
define void @test() {
  %nthreds1 = call i32 @omp_get_num_threads()
  call void @use(%nthreads1)

  call void @use(%nthreads1)
  ret void
}
```

HotColdSplitting `-hotcoldsplit`

- Split hot regions and cold regions
  - Extract cold regions to improve locality
- Example

[Hot Cold Splitting Optimization Pass In LLVM. A. Kumar. LLVM Developers' Meeting 2019]

```
extern void bar(int);
extern void __attribute__((cold)) sink();
void foo(int cond) {
  if (cond) {
    if (cond > 10)
      bar(0);
    else
      bar(1);
    sink();
  }
  bar(2);
}
```

Extract

```
extern void bar(int);
extern void __attribute__((cold)) sink();

void foo_cold(int cond) {
  if (cond > 10)
    bar(0);
  else
    bar(1);
  sink();
}

void foo(int cond) {
  if (cond) {
    foo_cold(cond);
  }
  bar(2);
}
```

Attributor

`-attributor`
`-attributor-cgscc`
`-attributor-enable={all,module,cgscc} -O{1,2,3,...}`

- Fixpoint iteration framework
  - Deduce *various* (>20 now) “attributes” aggressively and simultaneously
- Two versions
  - CGSCC pass and Module pass
- Example

```
define i32 @f(i32* %ptr, i32 %x) {
  %load = load i32, i32* %ptr
  %res = add i32 %load, %x
  ret i32 %res
}
```

```
define i32 @f(i32* nocapture nofree nonnull readonly align 4 dereferenceable(4) %ptr,
              i32 %x) #0 {
  %load = load i32, i32* %ptr, align 4
  %res = add i32 %load, %x
  ret i32 %res
}

attributes #0 = { argmemonly nofree nosync nounwind readonly willreturn }
```

[The Attributor: A Versatile Inter-procedural Fixpoint Iteration Framework, J. Doerfert, LLVM Developers’ Meeting 2019]

Inlining (in LLVM)

Stefanos Baziotis
NEC Corporation and University of Athens
users.uoa.gr/~sdi1600105/
stefanos.baziotis@gmail.com

Inlining

- Replaces a function call (site) with the body of the called function.

Inlining

- Replaces a function call (site) with the body of the called function.
- Inlining is a relatively simple transformation. It’s the decision of whether (and how much) to inline or not that is difficult.

Inlining

- Replaces a function call (site) with the body of the called function.
- Inlining is a relatively simple transformation. It’s the decision of whether (and how much) to inline or not that is difficult.
- Actually, it has been shown to be at least as hard as the Knapsack problem, so, NP-complete[1].

[1] Scheifler, R. W. 1977. An analysis of inline substitution for a structured programming language. Communications of the ACM, 20(9), 647--654

Inlining

- Replaces a function call (site) with the body of the called function.
- Inlining is a relatively simple transformation. It's the decision of whether (and how much) to inline or not that is difficult.
- Actually, it has been shown to be at least as hard as the Knapsack problem, so, NP-complete[1].
- For that reason, people have been using hand-written heuristics that "empirically work". Lately, Machine Learning is being used.

[1] Scheifler, R. W. 1977. An analysis of inline substitution for a structured programming language. Communications of the ACM, 20(9), 647--654

Inlining - Can We Always Inline ? No!

Usually, because we don’t have the function code:

- Other Modules / Compilation Units (LTO can help there)
- Shared Libraries
- Calls through function pointers (so, also virtual calls)
  - In reality, the compiler may inline some of the candidates in place[2,3].

[2] Compiler Confidential, Eric Brumer, GoingNative 2013

[3] Devirtualization in LLVM, P. Padlewski, LLVM Developers’ Meeting 2016

Inlining - Can We Always Inline ? No!

But also because of weird code structure:

- Recursive functions
  - Although tail recursion can be inlined.
  - Also, if at some point we can turn recursion into loops.

Inlining - Benefits

- Removes branching because of call.

Inlining - Benefits

- Removes branching because of call.
  - May help in (instruction cache) locality, for example if we inline a function in a loop.

```
150 void foo() {
151   for (...) {  // hot loop
152     bar();
153     ...
154   }
155 }
```

```
1 void bar() {
2   ...
3 }
```

Inlining - Benefits

- Removes branching because of call.
  - May help in (instruction cache) locality, for example if we inline a function in a loop.

```
150 void foo() {
151   for (...) {  // hot loop
152     bar();
153     ...
154   }
155 }
```

```
1 void bar() {
2   ...
3 }
```

Inlining - Benefits

- Removes branching because of call.
  - May help in (instruction cache) locality, for example if we inline a function in a loop.

```
150 void foo() {
151   for (...) {  // hot loop
152     bar();
153     ...
154   }
155 }
```

```
1 void bar() {
2   ...
3 }
```

Inlining - Benefits

- Removes branching because of call.
  - May help in (instruction cache) locality, for example if we inline a function in a loop.

```
150 void foo() {
151   for (...) {  // hot loop
152     bar();
153     ...
154   }
155 }
```


```
1 void bar() {
2   ...
3 }
```

Inlining - Benefits

- Removes branching because of call.
  - May help in (instruction cache) locality, for example if we inline a function in a loop.

```
150 void foo() {
151   for (...) {  // hot loop
152     bar();
153     ...
154   }
155 }
```

```
1 void bar() {
2   ...
3 }
```

Inlining - Benefits

- Removes branching because of call.
  - May help in (instruction cache) locality, for example if we inline a function in a loop.

```
150 void foo() {
151   for (...) {  // hot loop
152     bar();
153     ...
154   }
155 }
```

```
1 void bar() {
2   ...
3 }
```

Inlining - Benefits

- Removes branching because of call.
  - May help in (instruction cache) locality, for example if we inline a function in a loop.

```
150 void foo() {
151   for (...) {  // hot loop
152     bar();
153     ...
154   }
155 }
```

miss

```
1 void bar() {
2   ...
3 }
```

Inlining - Benefits

- Removes branching because of call.
  - May help in (instruction cache) locality, for example if we inline a function in a loop.
- Removes save / restore of registers, function prologue / epilogue etc.
  - Common heuristic: If the (actual) function code is less than two times the Call Instruction Sequence, inline it.

Inlining - Benefits

But ***most importantly: It is an enabling transformation!***

```
define internal i32 @callee(i32 %A, i32 %B) {
  %C = sdiv i32 %A, %B
  ret i32 %C
}

define i32 @caller() {
  %X = call i32 @callee(i32 10, i32 3)
  ret i32 %X
}
```

```
define i32 @caller() {
  ret i32 3
}
```

Inlining - Drawbacks

- Code Duplication
  - Analyze same code multiple times

Inlining - Drawbacks

- Code Duplication
  - Analyze same code multiple times
- Code Size Explosion
  - Executable Size Grows
  - Impacts the Instruction Cache

```
1  void bar() {
2    // big function
3  }
4
5  void foo() {
6    for (...) {  // hot loop
7      if (COND1)  // COND1 is rarely true
8        bar();
9      ...
10   }
11 }
```

Godbolt Snippet

Inlining - Drawbacks

- Code Duplication
  - Analyze same code multiple times
- Code Size Explosion
  - Executable Size Grows
  - Impacts the Instruction Cache

If this is latency-sensitive code, that may be a good decision!

```
1  void bar() {
2    // big function
3  }
4
5  void foo() {
6    for (...) {  // hot loop
7      if (COND1)  // COND1 is rarely true
8        bar();
9      ...
10   }
11 }
```

Godbolt Snippet

Inlining - Drawbacks

- Code Duplication
  - Analyze same code multiple times
- Code Size Explosion
  - Executable Size Grows
  - Impacts the Instruction Cache
- Increased Register Allocator Pressure
  - There’s no register save / restore
    - Live ranges of registers are extended
  - More loop invariants may be discovered
    - More registers to keep them

Inlining in LLVM - Place in the Pipeline

Because it is the most important enabling transformation, inlining happens early in the pipeline. And it is the main focus of it.

Inlining in LLVM - Pass Manager

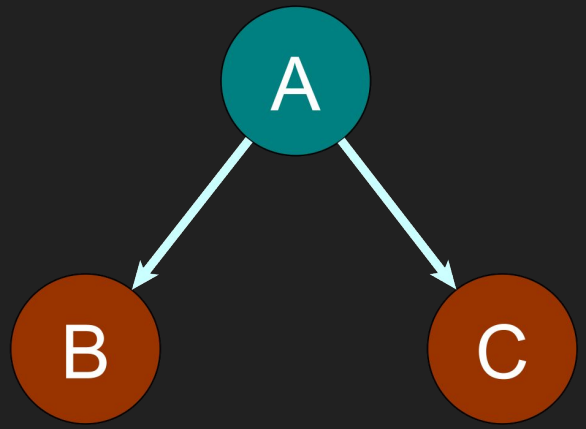
- Inlining is a Call-Graph SCC pass, which means it visits inlining candidates in a bottom-up SCC order.
  - First callees, then callers

Inlining in LLVM - Pass Manager

- Inlining is a Call-Graph SCC pass, which means it visits inlining candidates in a bottom-up SCC order.
  - First callees, then callers
- The Pass Manager interlaces function passes between the visits of the inliner to the functions.

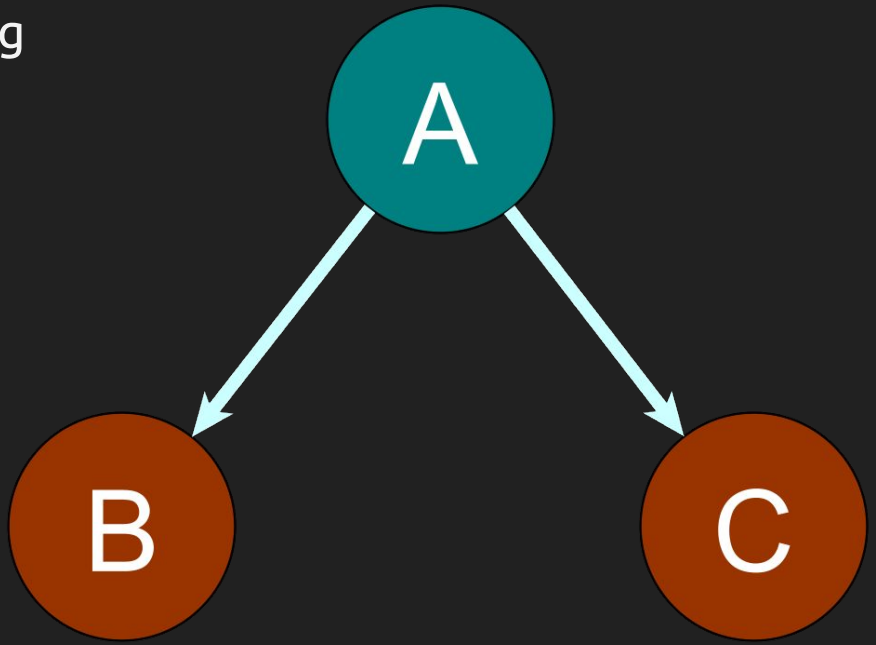
Inlining in LLVM - Example of Pass Ordering

```
1 int B() { return 2; }
2 int C() { return 1; }
3
4 void A() {
5   return B() + C();
6 }
```



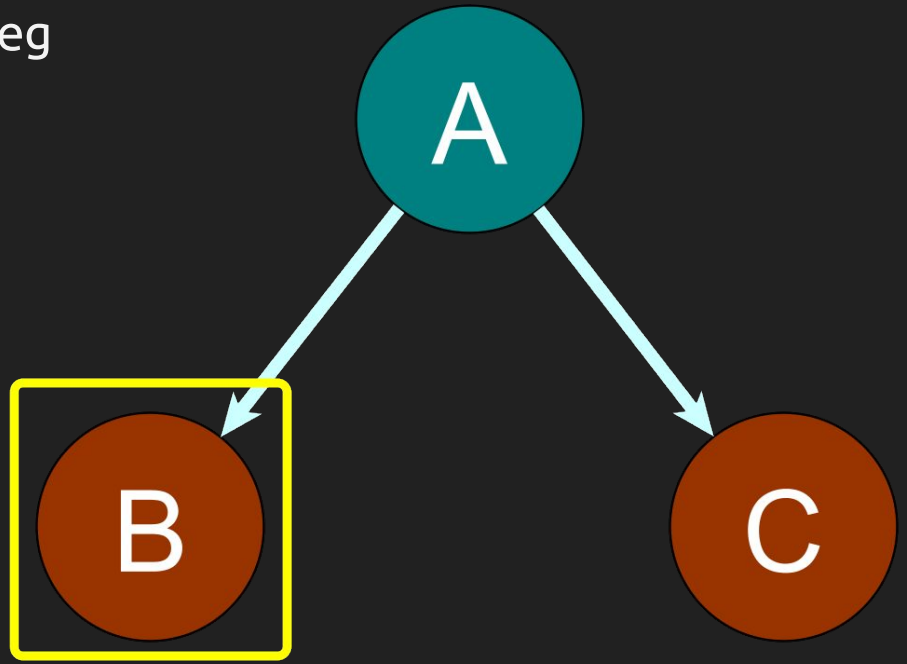
Inlining in LLVM - Example of Pass Ordering

```
opt -inline -mem2reg
```



Inlining in LLVM - Example of Pass Ordering

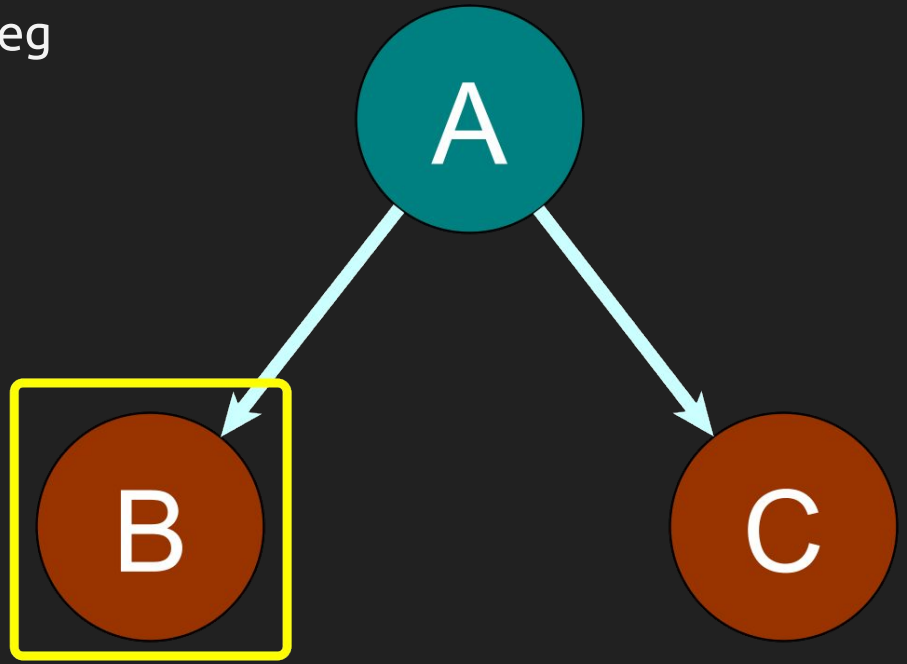
```
opt -inline -mem2reg
```



- Run inliner on `B()`

Inlining in LLVM - Example of Pass Ordering

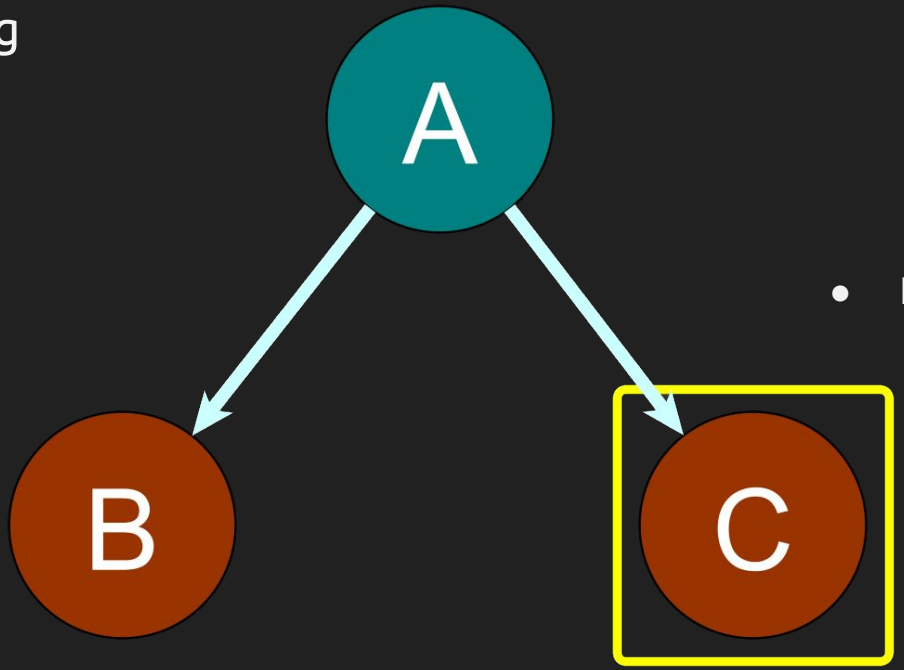
```
opt -inline -mem2reg
```



- Run inliner on `B()`
- Run mem2reg on `B()`

Inlining in LLVM - Example of Pass Ordering

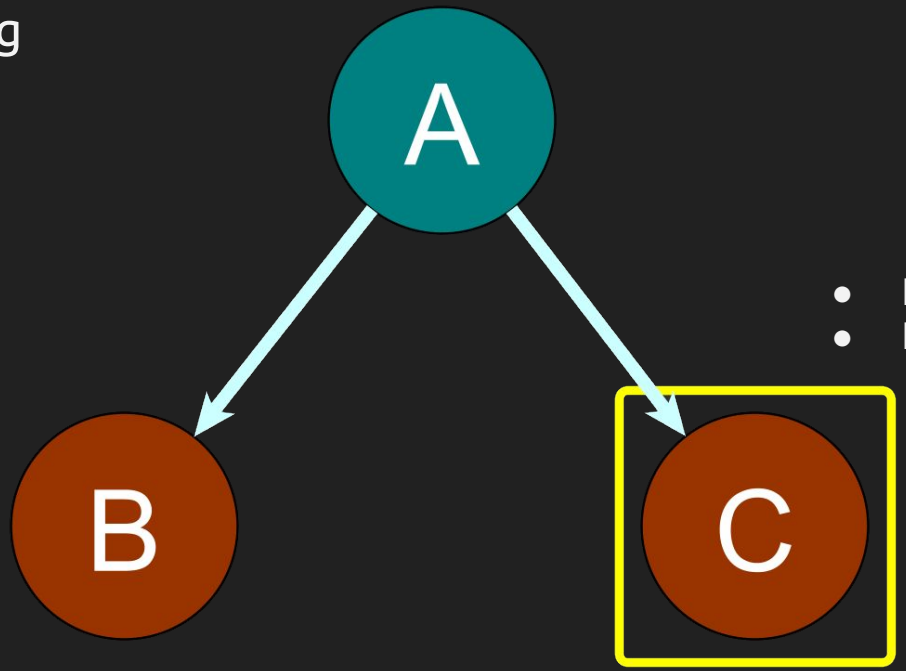
```
opt -inline -mem2reg
```



- Run inliner on `C()`

Inlining in LLVM - Example of Pass Ordering

```
opt -inline -mem2reg
```

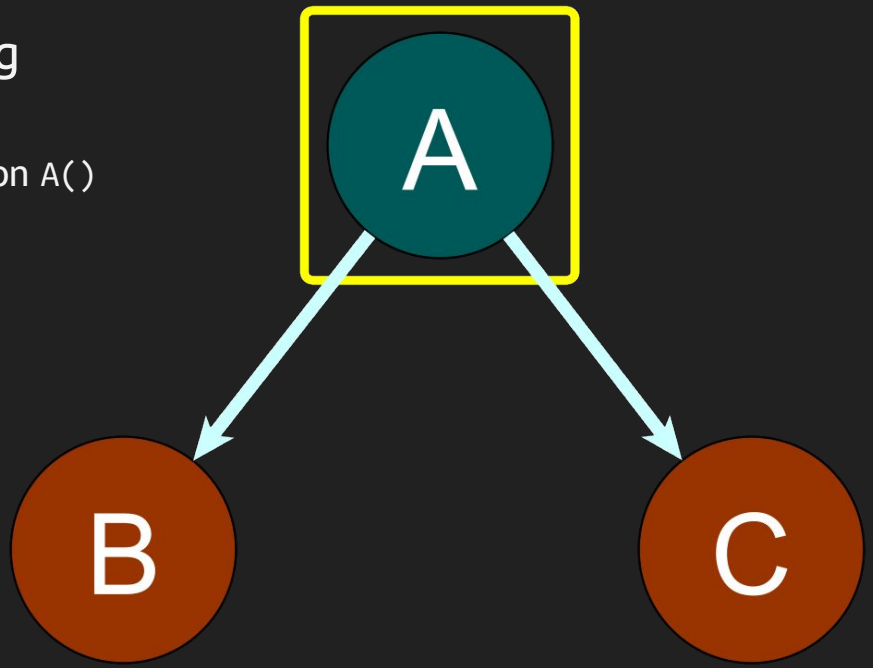


- Run inliner on `C()`
- Run mem2reg on `C()`

Inlining in LLVM - Example of Pass Ordering

```
opt -inline -mem2reg
```

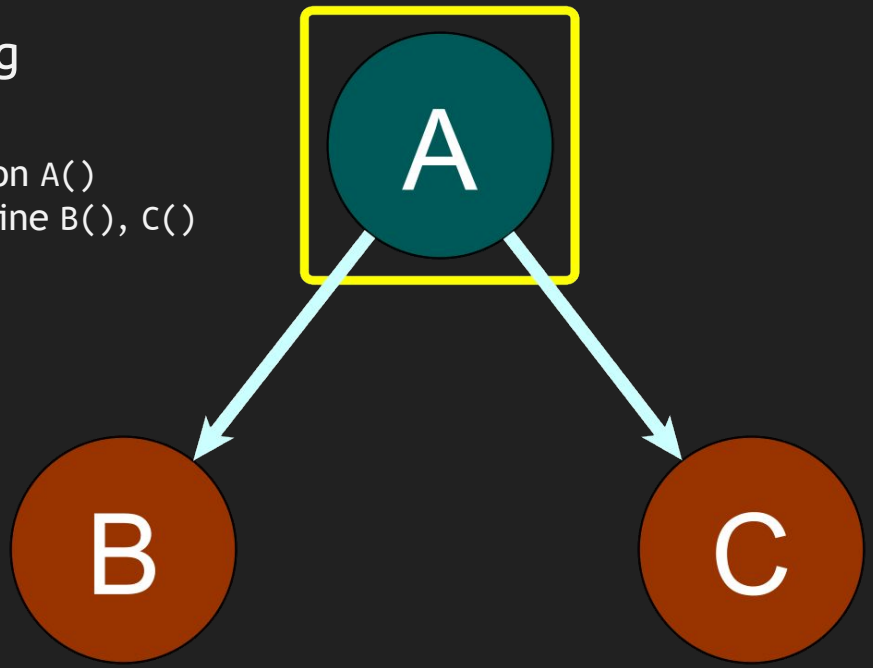
- Run inliner on `A()`



Inlining in LLVM - Example of Pass Ordering

```
opt -inline -mem2reg
```

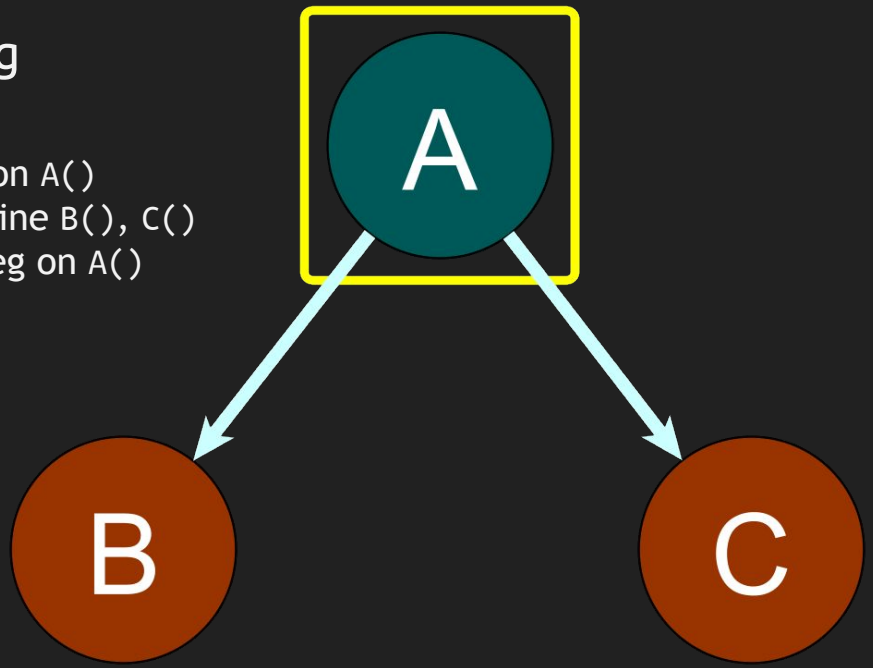
- Run inliner on `A()`
  - We inline `B()`, `C()`



Inlining in LLVM - Example of Pass Ordering

```
opt -inline -mem2reg
```

- Run inliner on `A()`
  - We inline `B()`, `C()`
- Run mem2reg on `A()`



Further Reading (in chronological order)

- Scheifler, R. W. 1977. An analysis of inline substitution for a structured programming language. Communications of the ACM, 20(9), 647--654
- W. W. Hwu and P. P. Chang, Inline Function Expansion for Compiling Realistic C Programs,Proc. ACM SIGPLAN 1989 Conf. Progr. Lang. Design and Implementation, pp. 246–257
- S. Richardson and M. Ganapathi. Interprocedural analysis versus procedure integration. Information Processing Letters, 3~(3), 137-142, August 1989
- Cooper, K.D., Hall, M.W. and Torczon, L. (1991), An experiment with inline substitution. Softw: Pract. Exper., 21: 581-601. doi:10.1002/spe.4380210604
- McFarling, S.: Procedure merging with instruction caches. In: Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, pp. 71–79 (June 1991)
- J. W. Davidson and A. M. Holler, "Subprogram inlining: a study of its effects on program execution time," in IEEE Transactions on Software Engineering, vol. 18, no. 2, pp. 89-102, Feb. 1992, doi: 10.1109/32.121752.
- K. Cooper, M. Hall and K. Kennedy, "Procedure cloning," in Proceedings of the 1992 International Conference on Computer Languages, Oakland, CA, USA, 1992 pp. 96,97,98,99,100,101,102,103,104,105. doi: 10.1109/ICCL.1992.185472
- Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-Mei W. Hwu. 1992. Profile-guided automatic inline expansion for C programs. Software: Practice and Experience , Vol. 22, 5 (1992), 349--369.
- Cooper, K.D., Hall, M.W., Torczon, L.: Unexpected side effects of inline substitution: a case study. ACM Lett. Program. Lang. Syst. 1(1) (March 1992)

Further Reading (in chronological order)

- Jagannathan, S., & Wright, A.K. (1996), “Flow-directed inlining”. PLDI '96.
- A. Ayers, R. Schooler and R. Gottlieb, "Aggressive inlining", SIGPLAN Not, vol. 32, no. 5, pp. 134-145, 1997
- R. Muth, S. Debray, “Partial Inlining”, Technical Summary
- Owen Kaser, C.R. Ramakrishnan, “Evaluating inlining techniques”, Computer Languages, Volume 24, Issue 2, 1998, Pages 55-72
- Keith D Cooper, Mary W Hall, Ken Kennedy, “A methodology for procedure cloning”, Computer Languages, Volume 19, Issue 2, 1993, Pages 105-117
- D. Detlefs and O. Agesen, 1999, “Inlining of virtual methods”, In European Conference on Object-Oriented Programming, Springer, 258– 277
- Simon L. Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler inliner. J. Funct. Program. 12, 4&5 (2002), 393–433
- M. Arnold, S. Fink, V. Sarkar and P. F. Sweeney, "A comparative study of static and profile-based heuristics for inlining", Proc. of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization, 2000
- Arnold, Matthew and Fink, Stephen and Sarkar, Vivek and Sweeney, Peter F., “A Comparative Study of Static and Profile-Based Heuristics for Inlining”, DYNAMO '00: Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization, January 2000, Pages 52–64

Further Reading (in chronological order)

- Kim Hazelwood and David Grove. 2003. Adaptive Online Context-sensitive Inlining. In Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '03). IEEE Computer Society, Washington, DC, USA, 253--264
- A. Monsifrot et al. 2002. A machine learning approach to automatic production of compiler heuristics. In Artificial Intelligence: Methodology, Systems, and Applications, LNCS 2443, D. Scott (Ed.). Springer, 41--50
- GCC Summit 2004, The GCC call graph module, Jan Hubicka
- Peng Zhao and J. N. Amaral, "Function outlining and partial inlining," 17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'05), Rio de Janeiro, RJ, Brazil, 2005, pp. 101-108
- Sameer Kulkarni and John Cavazos. 2012. Mitigating the Compiler Optimization Phase-ordering Problem Using Machine Learning. In Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12). ACM, New York, NY, USA, 147--162.
- Amir H Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A survey on compiler autotuning using machine learning. ACM Computing Surveys (CSUR) 51, 5 (2018), 96

Attributor

Attributor Overview

- Fixpoint iteration framework
  - Deduce *various* (>20 now) “attributes” aggressively and simultaneously
- Update states till fixpoint is reached
  - Dependencies between states are automatically caught by Attributor
- There are Module/CGSCC pass for both the old and new pass manager

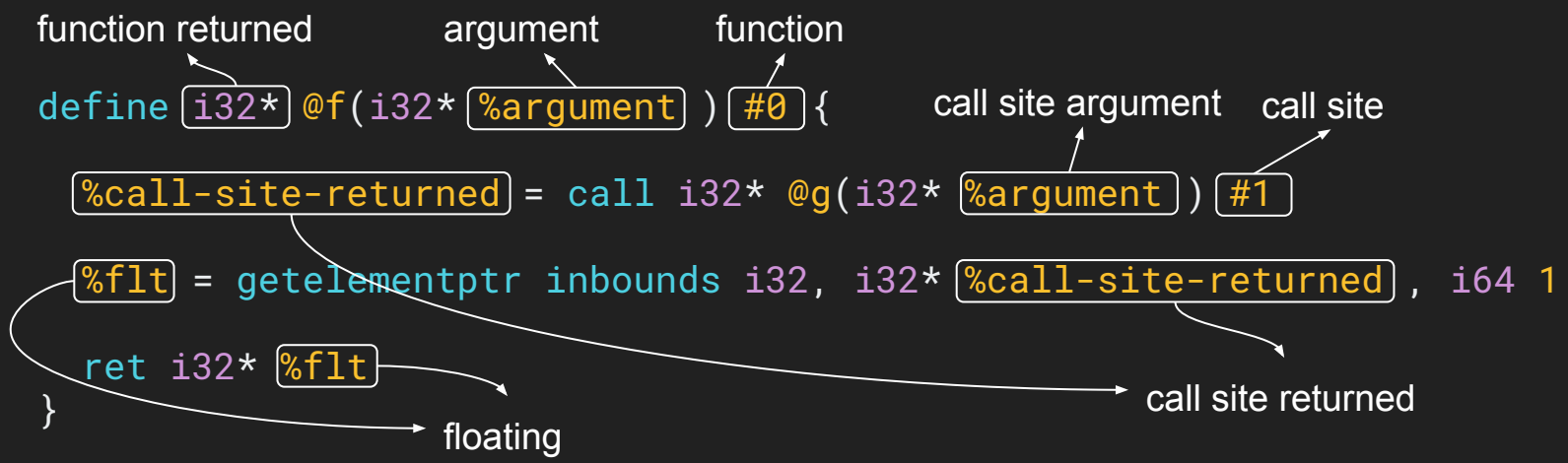
Why is it powerful?

- Attributor provides easy way to add new fixpoint analyses
- We can connect analyses with each other during fixpoint iteration
- Many existing IPO passes can be replaced by Attributor

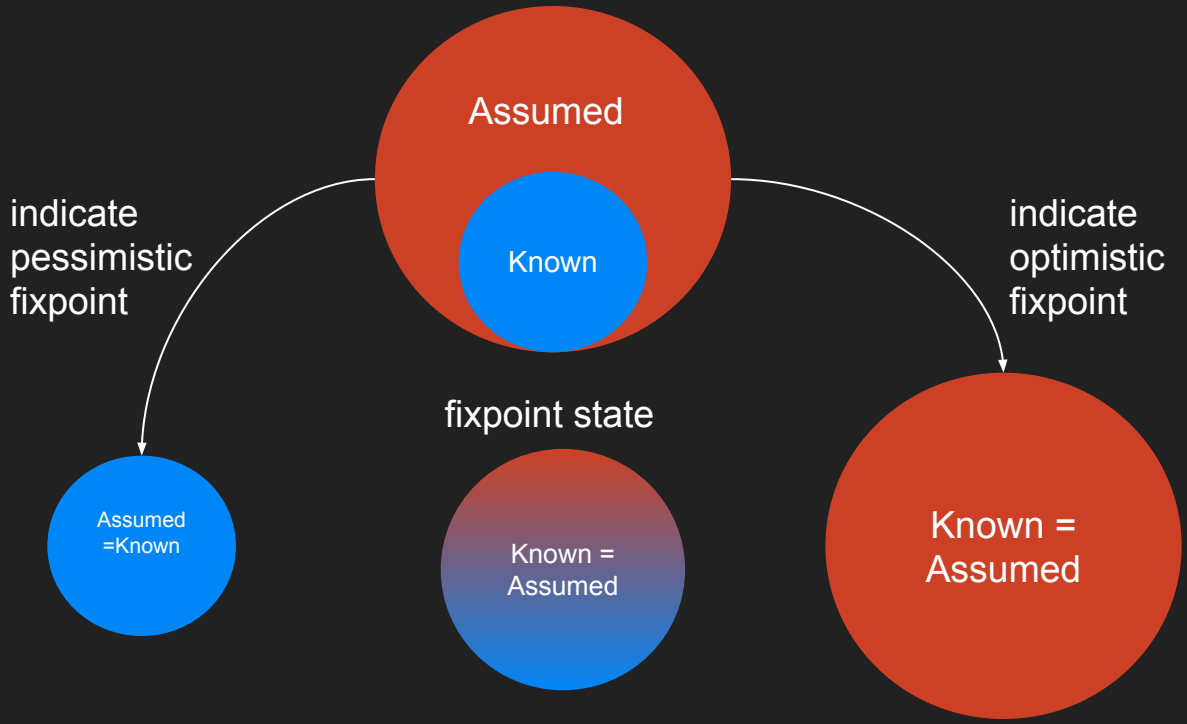
  - ✔ IPSCCP
  - ✔ Argument Promotion
  - ✔ Dead Argument Elimination
  - ✔ Infer Function Attrs
  - ✔ Prune EH

LLVM-IR Positions

- A class to specify positions in LLVM-IR https://llvm.org/doxygen/structllvm_1_1IRPosition.html

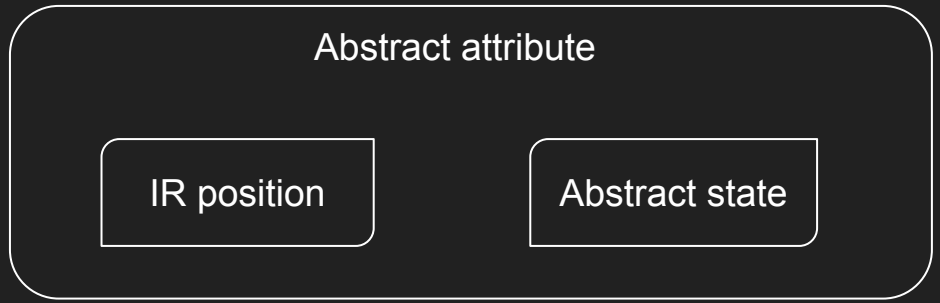


Abstract state



Abstract attribute

- What we call “attribute” here
  - Any stuff that describe properties of an IR position
  - Not only LLVM-IR attribute! (e.g. `nonnull`, `nocapture`, `nofree`, …)
- They are called “abstract attribute” in the code
  - `AbstractAttribute` class
  - Often abbreviated as AA



Abstract attribute: Example

- AAs that correspond to LLVM-IR attributes
  - `AANonNull ... nonnull`
  - `AANoCapture ... nocapture`
  - `AAAlign ... align`

- AAs that related to LLVM-IR attributes
  - `AAMemoryBehavior ... readnone, readonly, writeonly`
  - `AAMemoryLocation ... readnone, argmemonly, inaccessiblememory …`

- AAs that unrelated to any LLVM-IR attributes
  - `AAIsDead ...` Liveness Analysis
  - `AAValueSimplify ...` Value Simplification

Abstract attribute: Core methods

- `AbstractAttribute::initialize`
  - Initialize the state
- `AbstractAttribute::updateImpl`
  - Update the state
  - We can query states of some other AAs by `Attributor::getAAFor`
- `AbstractAttribute::manifest`
  - `Manifest the changes to the IR.`

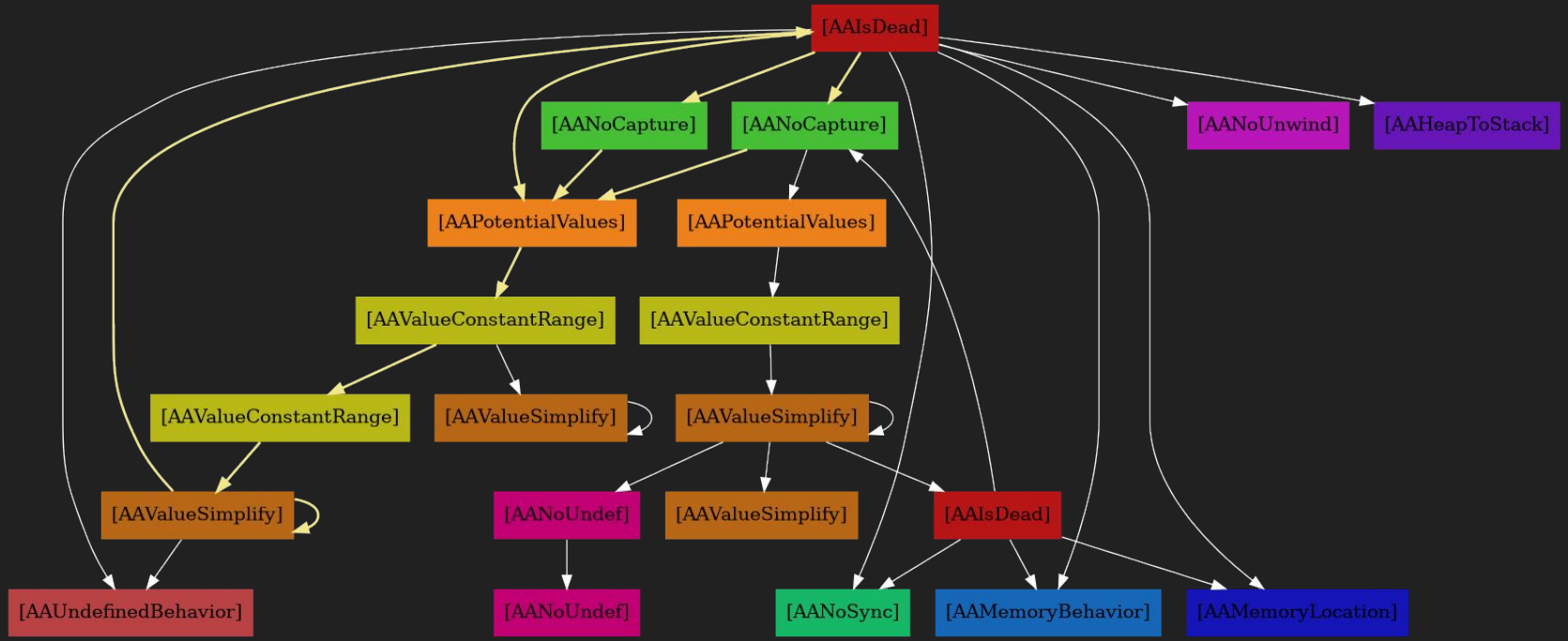
Update Function: Example

```
ChangeStatus AANonNullReturned::updateImpl(Attributor &A) {
  Function *F = getAnchorScope();
  auto Before = getState();
  auto& S = getState();

  for (Value *RetVal : /* Iterate all returned values of F in some way */)
    S &= A.getAAFor<AANonNull>(*this, IRPosition::value(RetVal));

  if (S == Before)
    return ChangeStatus::UNCHANGED;
  return ChangeStatus::CHANGED;
}
```

Clamp states for all returned values

```
declare nonnull i8* @foo()

define nonnull i8* @bar(i1 %c, i8* readnone %ptr) {
  br i1 %c, label %true, label %false

true:
  %q = getelementptr inbounds i8, i8* %ptr, i32 1
  ret i8* %q

false:
  %ret = call i8* @foo()
  ret i8* %ret
}
```

Dependency Graph



Phase of Attributor

Determine which kind of deduction or analysis we try to do

Update states till fixpoint is reached

Transform IR according to the results

Attributor Feature

- Performance related
  - Dependency type
- Utility for users
  - Helper classes for generic deduction
  - Helper functions for traversing assumed live uses, instructions, basicblocks...
  - Provides a uniform analysis pass query API
  - Selective seeding
  - Time traces

Attributor Feature

- Provides helper classes for generic deduction
  - All alive returned values → Function returned
  - All call sites → Function
  - All call site arguments → Function argument
- Example
  - `AAReturnedFromReturnedValues`

```
struct AANonNullReturned
     : AAReturnedFromReturnedValues<AANonNull, AANonNull> {
  /* We do not have to implement updateImpl */
};
```

Attributor Feature

- Provides abstract states for common situations
- Example
  - IncIntegerState
  - DecIntegerState
  - BitIntegerState
  - BooleanState

Attributor: Selective Seeding

- attributor-seed-allow-list

  Comma separated list of attribute names that are allowed to be seeded.

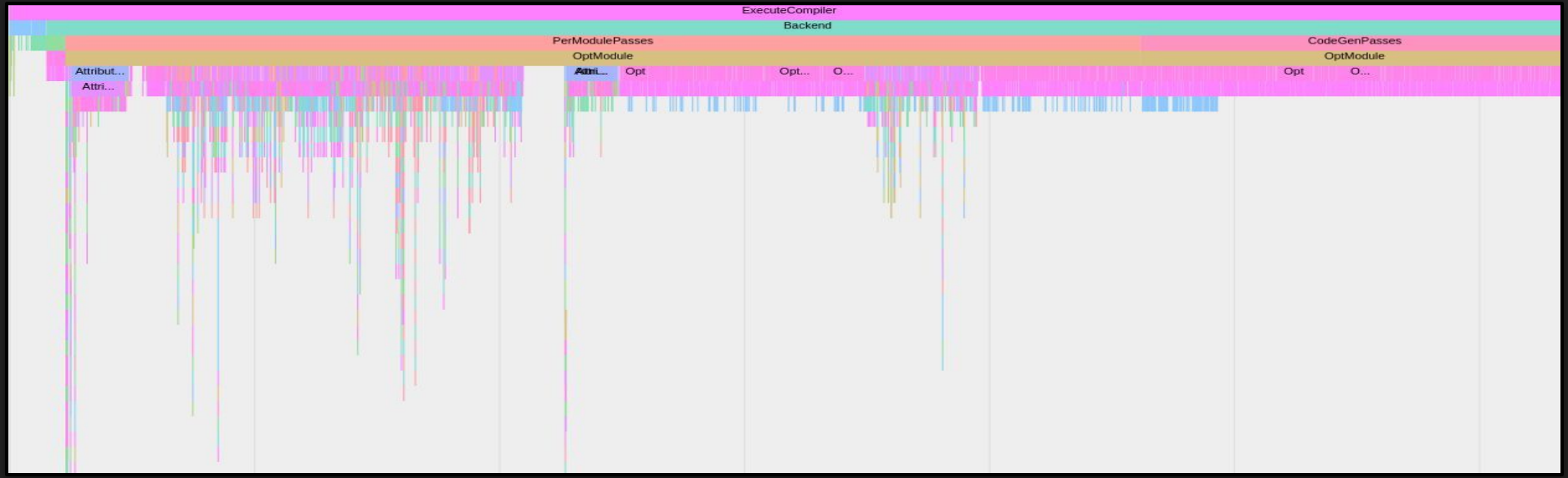
  --attributor-seed-allow-list=AANonNull

- attributor-function-seed-allow-list

  Comma separated list of function names that are allowed to be seeded.

  --attributor-seed-allow-list=foo

Attributor: Time Trace



Recap

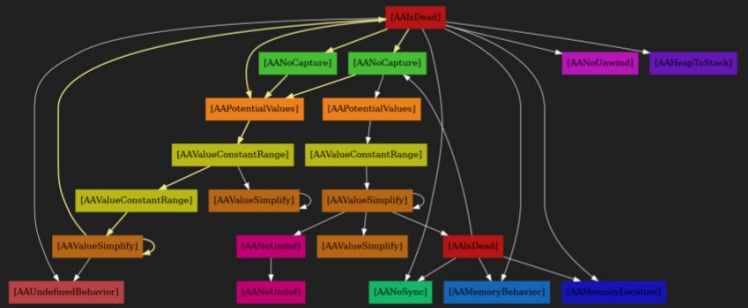
Existing IPO passes

Common IPO Pitfalls

- Scalability
- Complicated linkages
- Optimization pipeline, phase ordering
- Function pointer, different “kinds” of call sites, non-call site uses, …
- Variadic functions, complicated attributes (naked, byval, inreg, …)
- Keeping call graphs updated (for new and old pass managers)
  - `CallGraph` … old PM
  - `LazyCallGraph` … new PM

Recap - Attributor

Dependency Graph



Phase of Attributor

- **Seeding** — Determine which kind of deduction or analysis we try to do
- **Update** — Update states till fixpoint is reached
- **Manifest** — Transform IR according to the results

Recap

- Attributor technical talk & tutorial @ LLVM-Dev’19
- IPO panel @ LLVM-Dev’19
- IPO technical talk @ LLVM-Dev’20

Contact us if you are interested in any of this!

A Deep Dive into the Interprocedural Optimization Infrastructure